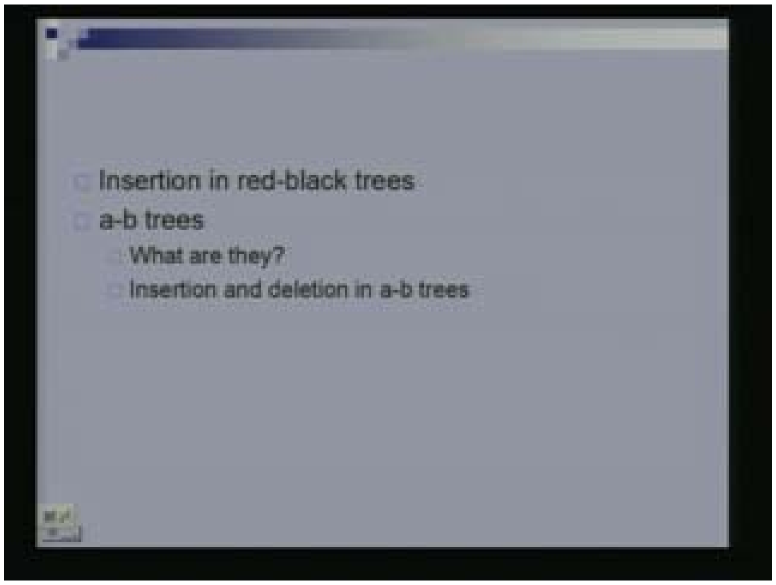
Data Structures and Algorithms
Dr. Naveen Garg
Department of Computer Science and Engineering
Indian Institute of Technology, Delhi
Lecture – 15

Insertion in Red Black Trees

In the last class we saw a red black tree, the correspondence of red black trees and 2-4 trees then we saw the deletion process in red black trees. This was the extensive process with 6 cases and so on. Today we are going to see how to insert the key in the red black tree also going to introduce the notion of an a-b tree. First we will define the a-b tree and then we are going to see the process of insertion and deletion in a-b tree.

(Refer Slide Time: 01:40)

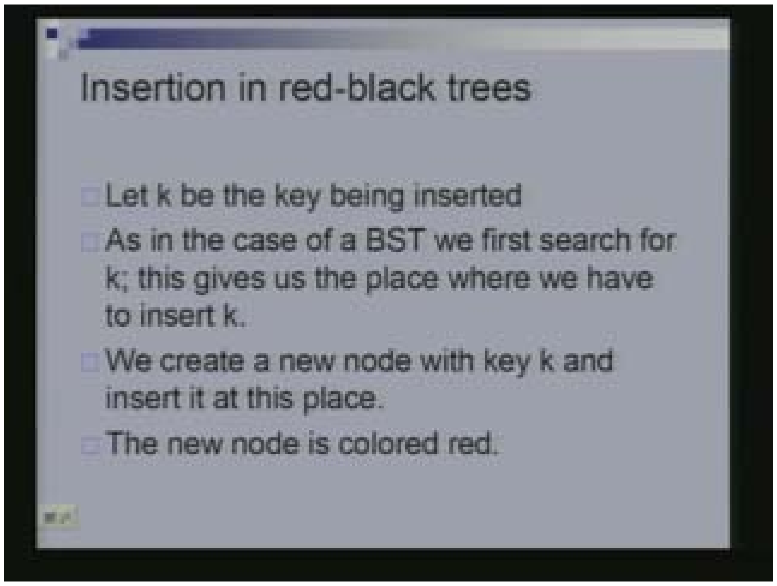


- Insertion in red-black trees
- a-b trees
  - What are they?
  - Insertion and deletion in a-b trees

We get to insertion. Suppose we are trying to insert a key k in to red black tree. After all a red black tree is a binary search tree. Since it is a binary search tree, first the insertion process would be like the binary search tree which means that we would try to find whether the key already exist in the tree. If it exists then we would not insert it. If it does not then we should be able to identify the place for the key to be inserted.

We create a node with that key, we put it at that location and we have to color this node. Because a red black tree differs from a binary search tree in the fact that each node is colored and this coloring obeys certain properties.

(Refer Slide Time: 01:50)



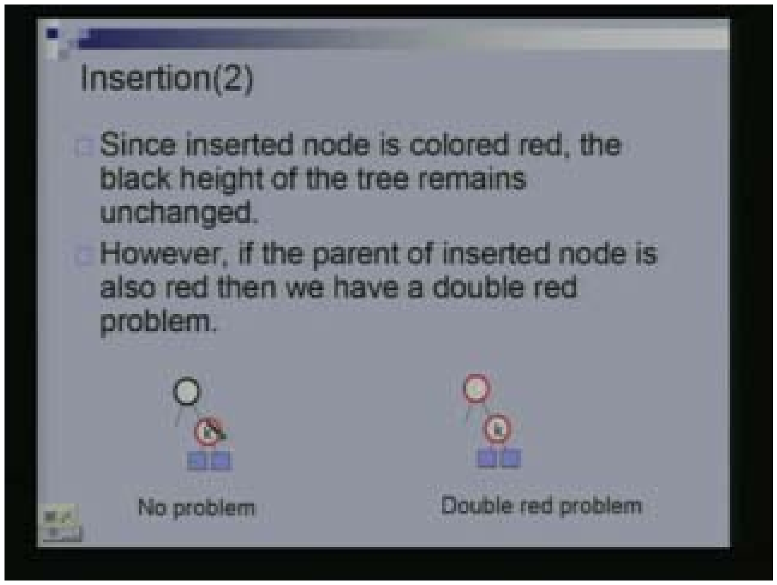
Insertion in red-black trees

- Let k be the key being inserted
- As in the case of a BST we first search for k; this gives us the place where we have to insert k.
- We create a new node with key k and insert it at this place.
- The new node is colored red.

Now we color this node and we are going to begin by coloring this node red. Let us say that in the slide given below the red colored node in the left side is the node that I inserted and it has a key k in it. Which means in my binary search tree I must have come up to the black colored node and gone right, because k was larger then this key and found that this was an external node.

Earlier in the binary search tree this was an external node. I decided to put my node to the right side of my black colored node and I color it red. I will create 2 external nodes which will be the children of this node. If the parent of this node is black then we have no problem because this node is colored red, the black height of the tree has not changed.

(Refer Slide Time: 02:31)



Insertion(2)

- Since inserted node is colored red, the black height of the tree remains unchanged.
- However, if the parent of inserted node is also red then we have a double red problem.

k
No problem
k
Double red problem

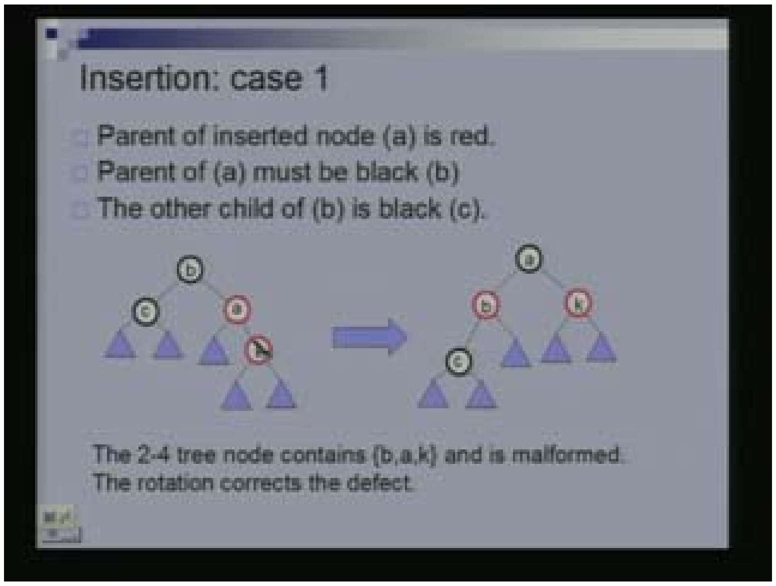
The black height of the external nodes of the children is the same as the black height of the external node which was sitting earlier at the location of the key and which was the same as the black height of the other external node in this tree. Which means that the black height of the 2 external nodes of the children is the same as the black height of the all the other external nodes. That property of the red black tree continues to hold, that is primarily because we have introduced a red node and not a black node.

The property that the black height of all the external nodes should be the same and it continues to hold. The problem could how ever be the double red colored node and that happens if the parent of the red colored node in the left side is red as it is shown in the right side of the above slide. The red colored node with key k inside it at the right side of the slide is the node I created. If its parent is red then we have the double red problem and we have to handle this problem.

Remember that in the case of deletion the problem was arising because the black height was changing and all along we were trying to take care of that problem. We never encountered a double red problem in the case of deletion. In the case of insertion how ever we will never have the problem of black heights not being uniform. The black heights of all the external nodes will be the same. No problem on that front but the problem will be one of a double red.

We have a double red problem. Let us see how to take care of this problem. Just concentrate on the left picture in the slide below. The k is the node that I am inserting and in the previous picture I had shown the 2 children who are external nodes and now I am just replace that by sub trees. You will see the reason for this in a short while.

(Refer Slide Time: 04:54)



Insertion: case 1

- Parent of inserted node (a) is red.
- Parent of (a) must be black (b)
- The other child of (b) is black (c).

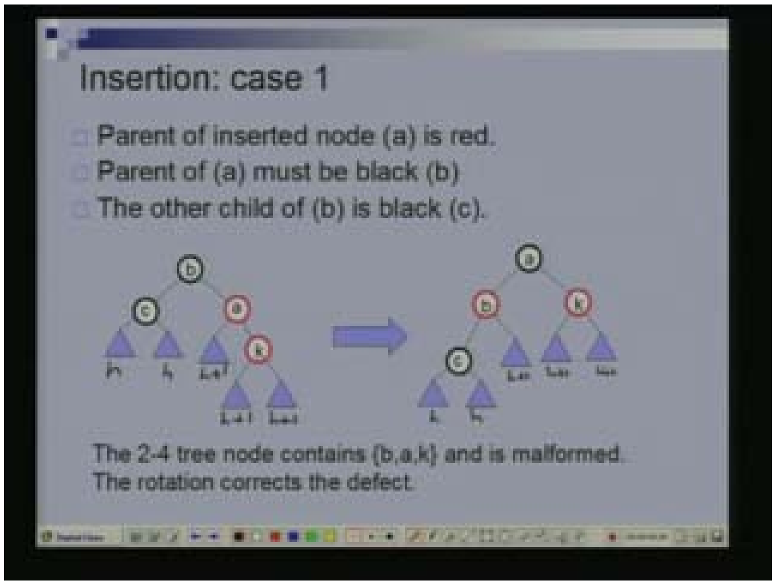
The 2-4 tree node contains (b,a,k) and is malformed.
The rotation corrects the defect.

We would have a double red problem if the parent of the node that we inserted is red. This parent which is node a is colored red. Clearly the parent of the node a must be a black. If the parent of this node was red then there was already a double red problem in my tree. So the node b must be a black and the first case that I am considering is when its other child is black which means that the sibling.

The k is the node that I am inserting, the sibling of its parent node a is black. That is the case I am considering. We just do a simple rotation. Note that a is larger then b and a is smaller then k so I can put a in the middle, b on the left and k on the right. I get a kind of a tree which is on the right side in the slide above. The a will be colored black, b and k would be colored red.

What is the black height of the tree on the left? The black height of the tree on the left is same as the black height of the tree on the right. If you take any external node then its black height was the black height of the external node plus one. And for the external nodes on the right, its height is still the same as the black height of those external nodes plus one.

(Refer Slide Time: 07:48)


Insertion: case 1

- Parent of inserted node (a) is red.
- Parent of (a) must be black (b)
- The other child of (b) is black (c).

The 2-4 tree node contains {b,a,k} and is malformed. The rotation corrects the defect.

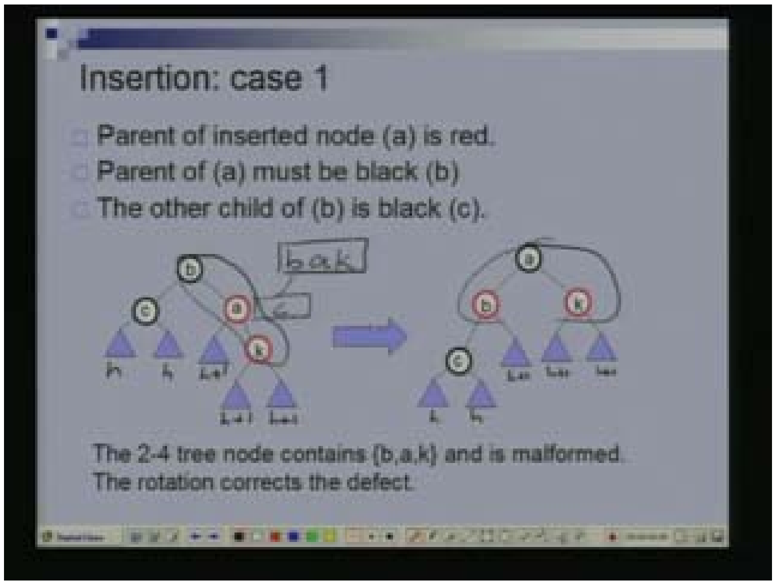
Would the black height of all of the nodes be the same? The black height of all the external nodes is indicated in the slide above. You can see that the black depth of all the external nodes. Anything below the node k will be (h+1) +1(on the node a) which will be h+2 and below the node c will be h+1(on the node c) +1(on the node b) which is h+2. So the black height problem not there at all, it is uniform.

Only thing we have to worry about this transformation is whether we have introduced a new double red problem, and we have not. We would have introduced a new double red problem if the root of any one of these 3 sub tree under node a and k was red. But if the root any one of these 3 sub tree was red then that means we had 2 double red problems and not 1 double red problem. If the external node under the node a was red and also the node a was red, then it is a double red problem. If the external node under k was red and also the node k was red, then it is a double red problem. We have more then one double red problem.

But we have introduced only one double red problem by inserting that node, so this cannot happen. In this case we are inserting k and for now we assume that the nodes under the node k are external nodes. In the next slide I will come to why I have drawn this 2 sub trees.

If I were to think of this as a 2-4 tree then that means that I have a node which has these 3 keys (a, b and k) in it. Earlier it had keys b and a in it, c was another node and k can be accommodated with b and a without any problem.

(Refer Slide Time: 09:56)



Insertion: case 1

- Parent of inserted node (a) is red.
- Parent of (a) must be black (b)
- The other child of (b) is black (c).

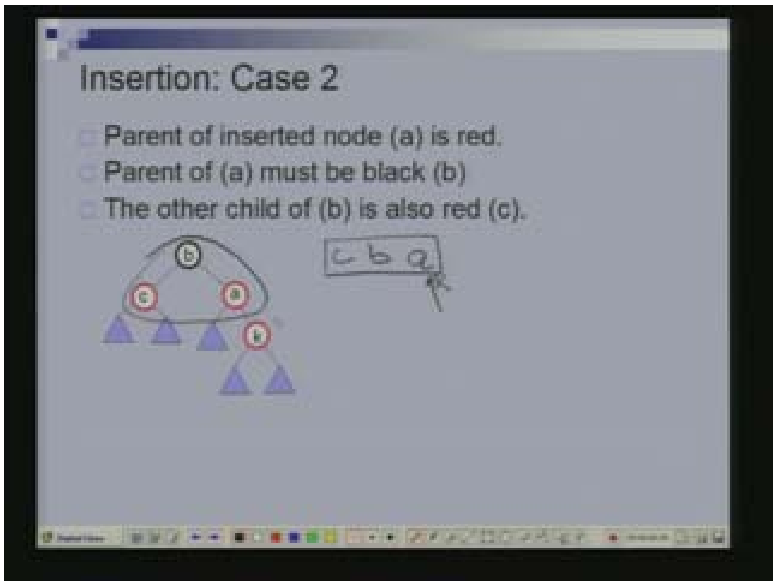
The 2-4 tree node contains {b,a,k} and is malformed.
The rotation corrects the defect.

This is what we said but we are getting a double red problem. Why is that? It is because this (b, a, k) is not formed in a right manner. If I had 3 keys in a 2-4 node then the middle key is the one which is to be set black, while it is not getting done in the left side tree on the above slide. Just that simple rotation takes care of this. So it is not that we are changing the 2-4 trees in the any manner. This (a, b, k in the left side) is the node corresponding to the 2-4 tree and these 3 goes together in to one 2-4 tree. These 3 nodes (b, a, k in the right side) still go together in the node of the 2-4 tree as before. It is just that we are reorganizing it, so that it is now in the form of our red black tree. So that was one case when the parent of the inserted node, has the sibling which is black. So the other cases when c is red and we will look at that now.

So this is the second case, the parent of the inserted node (a) is red and the other child of b is also red. We have this double red problem and we need to take care of this. What should I do now? Can I do which I did on the previous slide? Why not? Because then I would get a double red problem on the other side.

If I look at what is happening in the 2-4 tree that means in the 2-4 tree this (c, b, a) is the corresponding node of the 2-4 tree. Recall how I get the node of 2-4 trees, I take a black and look at all of its red children. So the nodes which are marked in the below slide is the corresponding node. It already has c, b and a in it and I am trying to bring in k in to this where clearly there is no space.

(Refer Slide Time: 11:48)

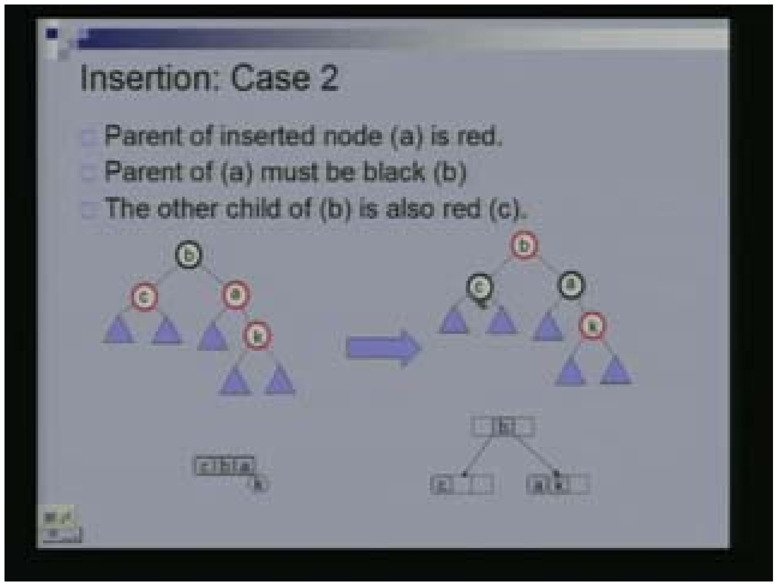


Insertion: Case 2

- Parent of inserted node (a) is red.
- Parent of (a) must be black (b)
- The other child of (b) is also red (c).

So we split it into 2. That is what we are going to see in the following slide.

(Refer Slide Time: 12:26)



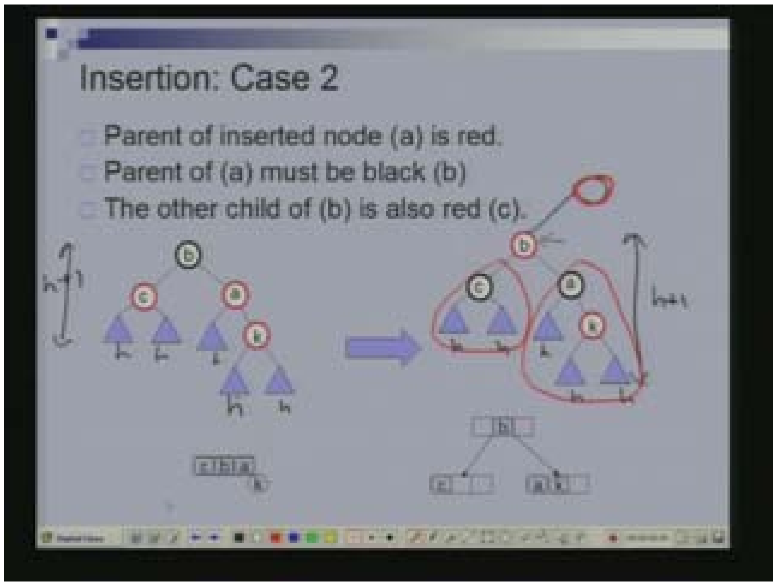
Insertion: Case 2

- Parent of inserted node (a) is red.
- Parent of (a) must be black (b)
- The other child of (b) is also red (c).

This is what the transformation we are going to do. No transformation but we just recolor the nodes and this corresponds to split and let us see why it happens. The left side tree in the above slide is the transformation that I have done. I have colored the node a and c to black. In the above picture (c, b, a) was sitting in one node and k is trying to come in. How do we split? We split with c on one side; a, k on the other side and then b goes up. Since c on one side that corresponds to a single black node, a and k on the other side corresponds to a and k on the right side of the tree and b goes up, so this is now trying to go up to the parent.

Does this take care of all problems? There are 2 issues. First have we created a black height problem? No, so let us say what should be the initial black height of all these sub trees?

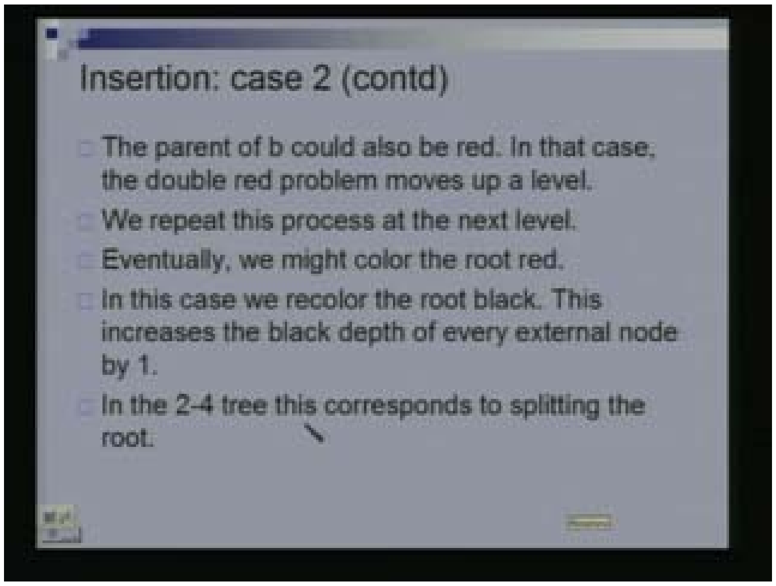
(Refer Slide Time: 14:37)


Insertion: Case 2

- Parent of inserted node (a) is red.
- Parent of (a) must be black (b)
- The other child of (b) is also red (c).

If the node below k is h, then all the black height of these entire sub trees which is on the left side would be h+1. This means the external nodes on the right side are all h and now the black height of these entire sub trees is h+1. So no problem, it was as before. But I have a red node b and its parent could have been a red. Thus I have a double red problem and this is the same thing. We have managed to move the double red problem one level up. And now you see why I had these sub trees hanging out of here. When this moves one level up, the rounded part in the above slide will be one sub tree hanging from the node b and the one which is marked on the right side would be the other sub tree hanging from here. This is the continuation, the parent of b could also be red and that case the double red problem moves up one level.

(Refer Slide Time: 14:53)



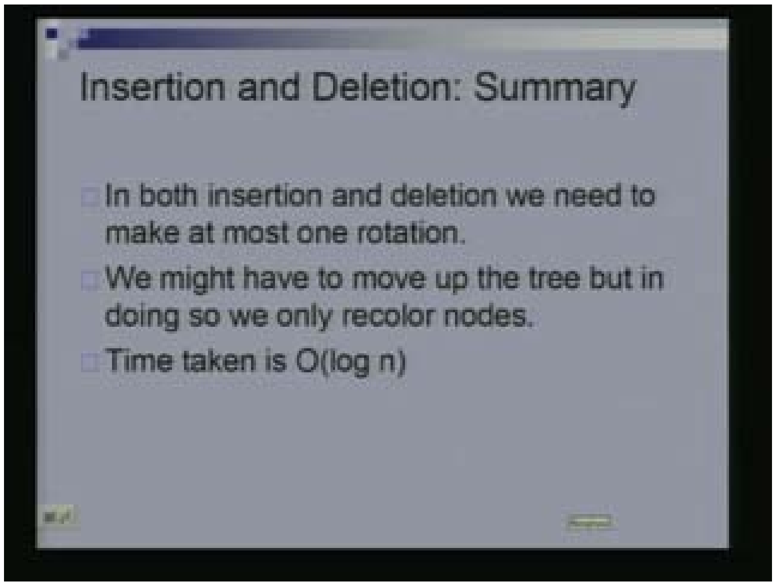
Insertion: case 2 (contd)

- The parent of b could also be red. In that case, the double red problem moves up a level.
- We repeat this process at the next level.
- Eventually, we might color the root red.
- In this case we recolor the root black. This increases the black depth of every external node by 1.
- In the 2-4 tree this corresponds to splitting the root.

We will repeat this process at the next level, we will consider the 2 cases. If we can by rotation take care of it we would have done it, if not then the double red problem will move even one level up and so on. Eventually we will end up coloring the root which was originally black as it was in a red black tree. We will end up coloring it red but if the root is colored red everything else is okay. That is the root is colored red. How do I take care of it? Just color it black again and that will increase the black depth of all the external nodes by one but it remains the same. We are not saying that the black depth of the all the external nodes should remain the same, we just say it should remain uniform.

The black depth of the one external node and the other external should be the same. If I color the root black, it will just affect all the external nodes by one. So there will be no problem. This essentially corresponds to moving all the way up and splitting the root. When we split the root in the case of a 2-4 tree, even then the height of 2-4 tree went up by one and so we are seeing that the height of the red black tree is correspondingly increasing by one when we do such thing. So again what we have seen in this insertion process is that either we have to do one rotation to take care of the problem and if we could not take care of the problem by one rotation then we have to move the problem up to the next level.

(Refer Slide Time: 16:31)

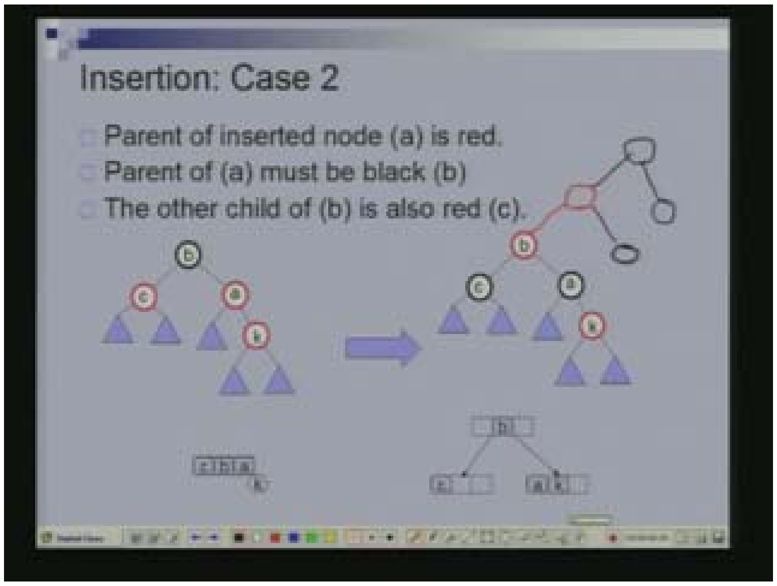


Insertion and Deletion: Summary

- In both insertion and deletion we need to make at most one rotation.
- We might have to move up the tree but in doing so we only recolor nodes.
- Time taken is O(log n)

But when we move the problem up to the next level we just did a recoloring of the nodes. Lets see, when we move the problem up to the next level in this case all I did was, change the reds to black and change the black to red. And now it is moved up to the next high level and may be if it has to moved up to further higher level it will just corresponds to recoloring of nodes.

(Refer Slide Time: 18:59)



Insertion: Case 2

- Parent of inserted node (a) is red.
- Parent of (a) must be black (b)
- The other child of (b) is also red (c).

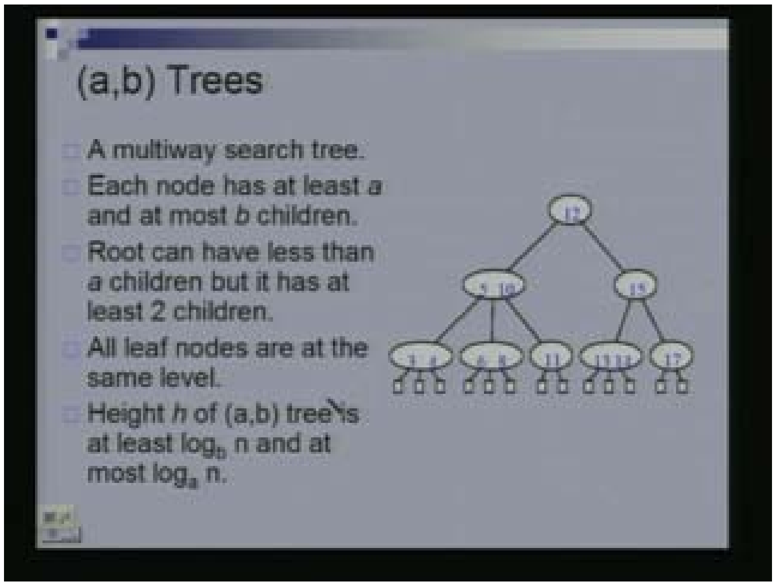
Suppose in the above slide, the parent of the red colored node b was red, then the other child of this has to be black, it cannot be red clearly. Other wise there was already a double red problem. But it is not that the one we are worried about. Its basically the node on the extreme right which we are worried about, whether it is red or black.

If we look at the previous case, we looked at the parent nodes sibling whether it is red or black. So it is the extreme right node whose color we are worried about and still this node can be red or black. So in insertion we just have to do one rotation. If we move up the tree, we just have to do recoloring. The same was happening in the case of deletion also and I had mentioned this very clearly that if we move up, we just had to do some recoloring. The moment you do one rotation the process ends, other wise it just does recoloring. This is what it makes the process very fast because recoloring is just one bit of information really in each node. The 1 or 0 will just tell you whether it is red or black. You just need to quickly change those bits if you are moving up the tree.

Rotation is slightly more expensive because it requires some pointer changes, 6 or 7 pointers have to be changed. And this is why the insertion and deletion in red black tree is faster than in the case of AVL trees. In AVL trees recall that we have to do more then one rotation. We did a rotation then we moved up, perhaps you have to do another rotation and so on. Although for both of the data structures, the worst case time for insertion and deletion is log n. Because even there you were doing only log n rotations, but the constant behind that log n are much larger in the case of AVL tree than in the case of the red black tree.

So even with in the log n, this would be a faster process for both insertion and deletion than in the case of an AVL tree. (Refer Slide Time: 16:31) That is all we wanted to discuss about the red black trees. We looked at search, insert and delete. All of them take log n time, you can also think of other operation like if I say find the minimum element in a red black tree. How much time do you think its going to take? Minimum means just keep going left. So time is height, height is log n then its just log n. You can do all such kind of operations in log n time. Most of those operations are not changing the tree. It is much easier, the 2 operations where we changed the tree are insert and delete but we seen that you can still take care of them in log n time.

(Refer Slide Time: 23:49)



(a,b) Trees

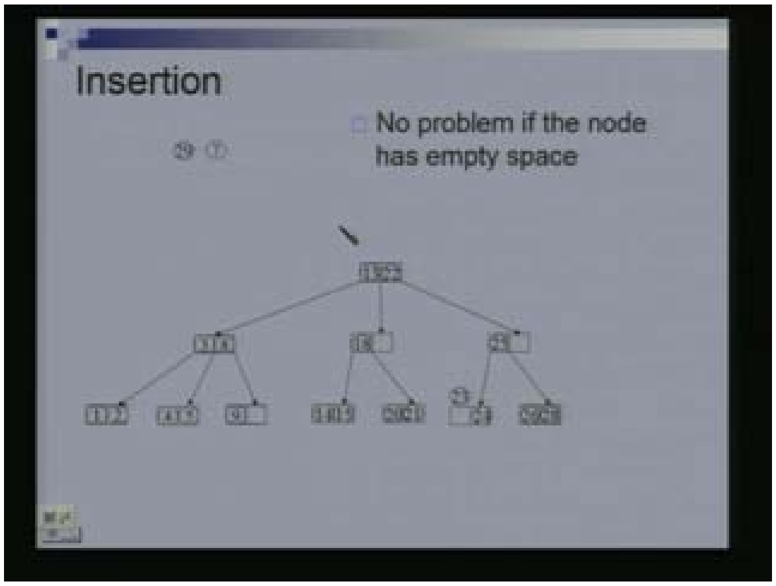
- A multiway search tree.
- Each node has at least *a* and at most *b* children.
- Root can have less than *a* children but it has at least 2 children.
- All leaf nodes are at the same level.
- Height *h* of (a,b) tree is at least $\log_b n$ and at most $\log_a n$.

We will come to this notion of a-b trees and this is a generalized idea of 2-4 trees.

What is an a-b tree? I have drawn an a-b tree and actually this is the same picture that I used for the 2-4 trees if you remember. An a-b tree is a multi-way search tree. Each node has at least a and at most b children. When a is 2 and b is 4 then you get 2-4 tree. If it has at least a children and at most b children then how many keys are there inside? a-1 to b-1. What do you mean by b-a+1? If it has a children then it has a-1 keys, if it has b children then it has b-1 keys. So the number of keys is between a-1 and b-1. The one node which does not satisfy this property is the root node. The root node can have only 2 children. If a is 3 or 7 or some other thing then its not that the root also should have at least 3 or 7 children. Root is out of this definition, so root can have only 2 children. Root has at least 2 and at most b children. For the root the requirement is from 2 to b and we will see what is the need for this requirement. Again all leaf nodes are at the same level.

What is the height of an a-b tree? We have seen this before, $\log_b n$ is the minimum height and $\log_a n$ is the maximum height. You can context this to the little bit because the root has the only 2 children. When the root has 2 children every one else has a children, so there would be a plus one in that. (Refer Slide Time: 24:15). But this ($\log_a n, \log_b n$) would be the roughly the bounds. As you can see in the above slide, every node has at least 2 and at most 3 children, so this is an example of 2-3 tree. We can talk of 2-3 tree, 2-4 tree, 2-5 trees and so on, for any choice of a and b. I will correct the statement as this discussion proceeds. It is not any choice of a and b, we will see what are our requirements on the relation between a and b. This will not work on any choice of a and b, but for now we will just assume it as any choice of a and b.

(Refer Slide Time: 25:04)



Insertion

- No problem if the node has empty space

In insertion so as you can imagine this is going to be essentially a repetition of what we did for 2-4 trees and it is with small modifications. I am trying to insert the key 21. As it is a multi way search tree, I will find a position where this has to go. The 21 does not go between 13 and 22 so come down. It is more then 18 so go right and it can fit in to that position next to 20. No problem if the node has an empty space. Similarly for 23, as it is less than 25 it comes to this space and in a node we are just keeping in order. So 24 will make way and 23 will come at the place.

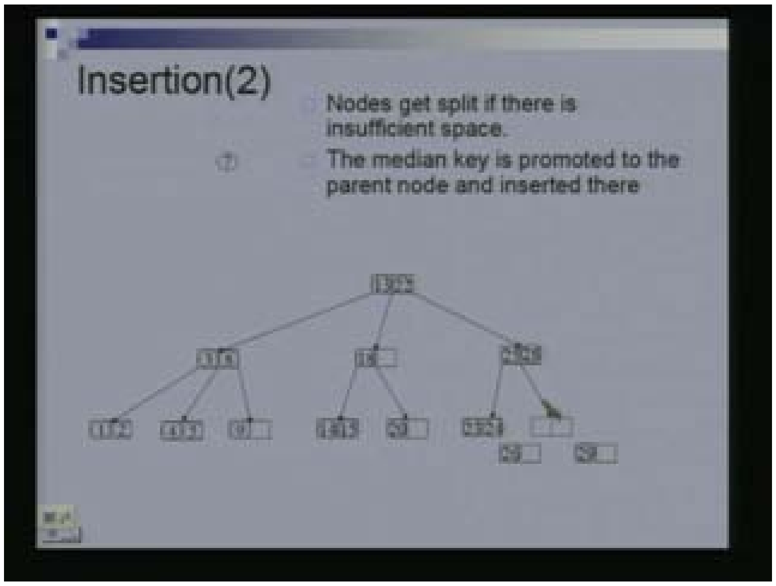
If there is no space in the node that we are trying to put the key in, then the node gets split. Let us see 29, 29 compared with 22 and it is more than 22 so it goes right, more than 25 and again goes right. I am looking at the 2-3 tree in this picture. We are talking of a-b but I did not want to make a and b very large because it would not fit on the slide. So I am just looking at a=2 and b=3. And the concepts are the same. For an a-b tree all the leaves have to be at the same height as in the case of 2-4 tree.

This is essentially 2-3 tree. Basically 2-3 tree means each node has between 2 and 3 children which means that each node can have 1 key or 2 keys only which is why each of those has been made with space for 2 keys. Each could have 1key or 2 keys.

This can have only 2 keys but now I am coming with another one that is the third one. If there is insufficient space then split the node. I am going to split this node and the median key is promoted to the parent. Thus 28 is the median of these 3, so 28 will get promoted to the parent. (Refer Slide Time: 27:54) So I split 26 goes down 29 goes down and 28 goes up. And the lines disappear and these become the children of this.

The split can cascade and we will see the example of that. When I am trying to insert 7 I will compare 7 with the first key. The 7 is less than 13 so I will go left, 7 lies between 3 and 8. So I take the middle path, come down and 7 tries to come to this node except this node does not have enough space.

(Refer Slide Time: 27:54)



Insertion(2)

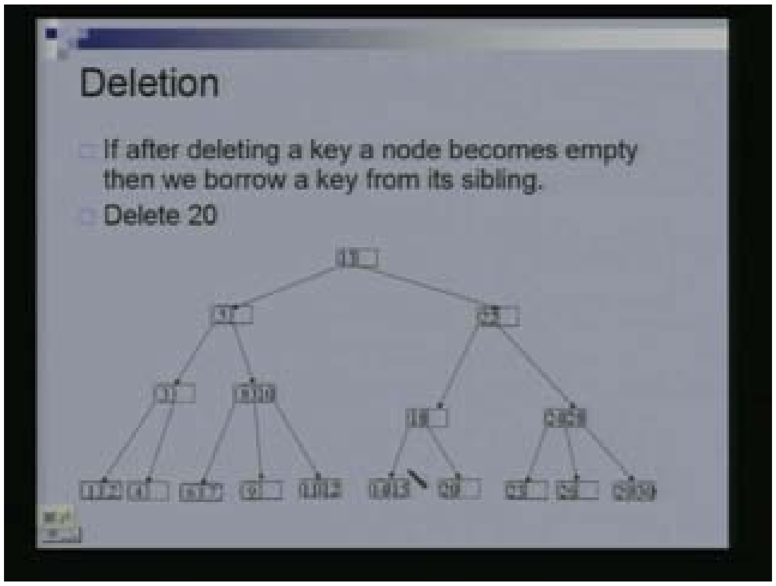
- Nodes get split if there is insufficient space.
- The median key is promoted to the parent node and inserted there

This node gets split in to 2. The 4 will go down, 7 will go to the adjacent node and median is 5 so it will go up. As nothing is left here because we just remove this node and we will remove it shortly. (Refer Slide Time: 28:51) This 5 is trying to go in to this (3, 8) node, except there is no space here either and once again this gets split in to 2. The 3 goes to the upper node and 8 to the adjacent node and the median 5 get promoted to the parent.

It is same as in the red black tree, we have not done anything which is different from the red black tree. But what I am pointing out through this is that we need not have a 2-4 tree, we could also have 2-3 tree. In the red black tree every node had space for 3 keys, even if every node had space for 2 keys we can still make it work. That is what happens, 3 and 8 get splitted and 5 goes to the top.

Once again 5 is trying to enter here, but there is no space for 5. First let us just reorganize this. These 4 children have to be children of these 2 nodes (3 and 8). The 2 left will go to the left of 3 and 2 right would go to its right. Let us take care of this node (5). The 5 try to go between 13 and 22, but not able to go. So this gets split in to 2 and 1 key gets promoted to the parent except there is no parent this is the root so we create a new root and we would go like that. (Refer Slide Time: 30:00) These are the 4 children (3, 8, 18, 25 28) they would have to become children of this two nodes (5 and 22). So 2 left most go to this (5) and 2 right most will go to this (22) and the new root (13) will have to this 2 children (5 and 22).

(Refer Slide Time: 35:47)



Deletion

- If after deleting a key a node becomes empty then we borrow a key from its sibling.
- Delete 20

Exactly the same thing would happen for any a-b tree. I am trying to insert, if there is space put it there, if there is no space split and move the median up. The median might not be unique like in the case of the 2-4 tree, there were 4 keys there. The median could be the second element or third element. We can not insert in to an empty node. What is this statement? There would be no empty node.

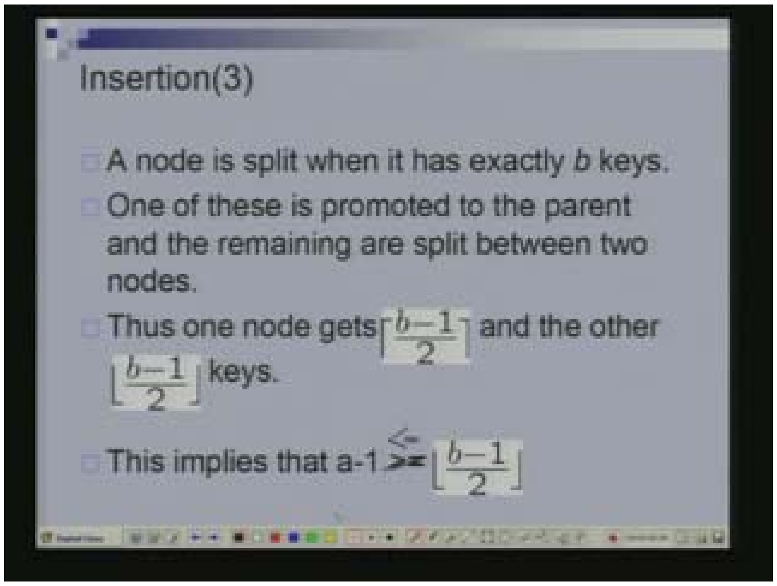
At least a children, at most b children mean every node has at least a-1keys and at most b-1keys. So this is the property. We can also rephrase it in this way. Every node has at least a-1keys and at most b-1keys. If your tree had less than a-1keys in it, then all of them would basically sit in the root. This property is not true for the root, root can have as small as 1 key only. Because it can have as small as 2 children. So root can have only 1key.

We are saying every node has at least a children and at most b children which mean that every node has at least a-1keys and at most b-1keys. This property of at least a children and at most b children does not apply to the root node. The root node has at least 2 and at most b. The root node has at least 1key and at most b-1keys. Why we said that this property does not apply to the root, because when we went in this manner and inserted and we ended up splitting the root node in to 2 and we created the new root node then this new root has only 2 children. We have to permit the root to have only 2 children. This is why this requirement, the root can have as small as 2 children. If we insist the root have to have at least a children then we might not be able to do this at all.

We are going back from 2-3 trees to a-b trees. So in a-b trees we said, we are trying to insert in a node if it has space, we put it there. How much space does an a-b tree node have? Space for b-1keys, if it has space we will put it. If it does not have space, what is that mean? They were already b-1keys there and I am trying to insert one more key, b^{th} key. Then we will split in to 2, one of the keys will go up and from the remaining b-1keys, half will go on one side and half will go on other side. That is exactly what is being said here. A node is split when it has exactly b keys. Why exactly b? Because earlier it had b-1, I was trying to put in one more there, then it means it has exactly b.

One of these is promoted to the parent and the remaining are split in to 2. What is the remaining? b-1, the b-1is getting split in to 2. One part gets $\left\lceil \frac{b-1}{2} \right\rceil$and the other part gets$\left\lfloor \frac{b-1}{2} \right\rfloor$. The$\frac{b-1}{2}$, if it is an integer then both of these are the same. If it is not an integer, like in the case of a 2-4 tree. The b is 4 in the case of a 2-4 tree. You had $\frac{4-1}{2}$ which is 1.5, which means one side was getting 2 keys and the other side was getting 1 key. Thus 1.5 rounded up is 2 and 1.5 rounded down is 1. One was getting 2 and the other 1. So one node gets $\left\lceil \frac{b-1}{2} \right\rceil$this many keys and the other gets $\left\lfloor \frac{b-1}{2} \right\rfloor$this many keys. But after the split, these 2 nodes are valid nodes of the key. So one node is getting so many keys $\left\lfloor \frac{b-1}{2} \right\rfloor$which means that a-1this quantity should be less then or equal to$\frac{b-1}{2}$.

(Refer Slide Time: 36:16)



Insertion(3)

- A node is split when it has exactly *b* keys.
- One of these is promoted to the parent and the remaining are split between two nodes.
- Thus one node gets $\left\lceil \frac{b-1}{2} \right\rceil$ and the other $\left\lfloor \frac{b-1}{2} \right\rfloor$ keys.
- This implies that a-1 $\overset{<=}{\cancel{>=}}$ $\left\lfloor \frac{b-1}{2} \right\rfloor$

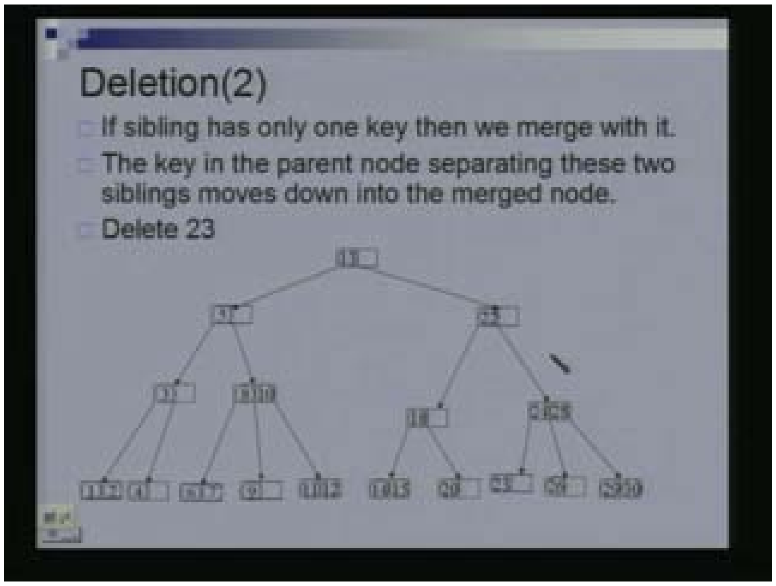
I have corrected in the above slide which is given below. a-1<= $\left\lfloor \frac{b-1}{2} \right\rfloor$ Why this is coming, because I am creating a node with $\frac{b-1}{2}$ keys which means I have a requirement that every node has at least a-1keys in it. This quantity should be greater than or equal to a-1. Otherwise if this was less than a-1, then there would be a problem because this node that I am creating is not a valid node.

Let us look at the deletion. Deletion again is the same as the case of a 2-4tree. The simple case is when I am deleting a certain node and there are more than one node in that. For instance if I were deleting 12 and there is nothing to be done, just delete 12 and still there is one key left. Suppose if I was deleting 20, then the problem would be that when I am removing 20, this node becomes empty and node has to have at least 1key and at most 2 keys. So it has to have at least one.

If the node becomes empty then I first try to borrow a key from its sibling. The sibling of 20 is 14 and 15. I will try to borrow one from that and recall the way we borrowed was one key went up and the other one came down. That is exactly 20 disappears and 15 goes up and 18 comes down. This is valid 2-3 tree.

What happens if the sibling has only one key? For instance if I was trying to delete 23, 23 goes away. I try to borrow one key from its sibling, 26 is the only sibling I can borrow from. I cannot borrow from the next one and you remember the reason for this. I am trying to borrow it from 26 but this has only 1key. If I borrow from 26, this becomes empty. In this case of a 2-4 tree we merge. First try to borrow if not possible then merge.

(Refer Slide Time: 38:33)



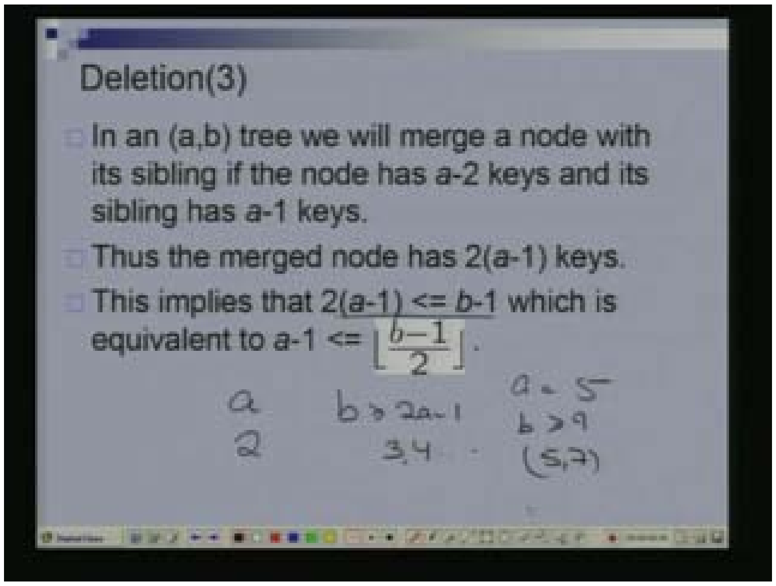
Deletion(2)

- If sibling has only one key then we merge with it.
- The key in the parent node separating these two siblings moves down into the merged node.
- Delete 23

We will merge this and we will create a new node. And this will get the key 26 and one key will come down from the parent. The 24 will come down and 26 will come from here and this will become the merge node. Thus only one key left there, so it goes to the left and these become the 2 children of this node. We also saw that this process could go up and up. Because what we had effectively done is that we have removed one key from this 28th node. But if this node had only one key then it would have become empty.

If this 28 was not there then this would have become empty. Then we would have tried to borrow one from 18, but we cannot borrow because there is only one key. This 18 would have merged which means this 22 would have come down and we would have created one node with 18 and 22. Which means I would have deleted something from 22 but if I am deleting something from here this becomes empty. So again it will merge and in this manner eventually the root has to be removed and the height of the tree reduces by one. We have seen this example from the case of the 2-4 tree, exactly the same thing is happening. Let us continue. In an a-b tree we will merge a node with its sibling. Now we are coming back to the a-b tree. When do I merge a node with its sibling?

(Refer Slide Time: 45:36)



Deletion(3)

- In an (a,b) tree we will merge a node with its sibling if the node has *a*-2 keys and its sibling has *a*-1 keys.
- Thus the merged node has 2(*a*-1) keys.
- This implies that 2(*a*-1) <= *b*-1 which is equivalent to *a*-1 <= $\left\lfloor \frac{b-1}{2} \right\rfloor$.

a b ≥ 2a-1 a = 5
2 3,4 . b ≥ 9
(5,7)

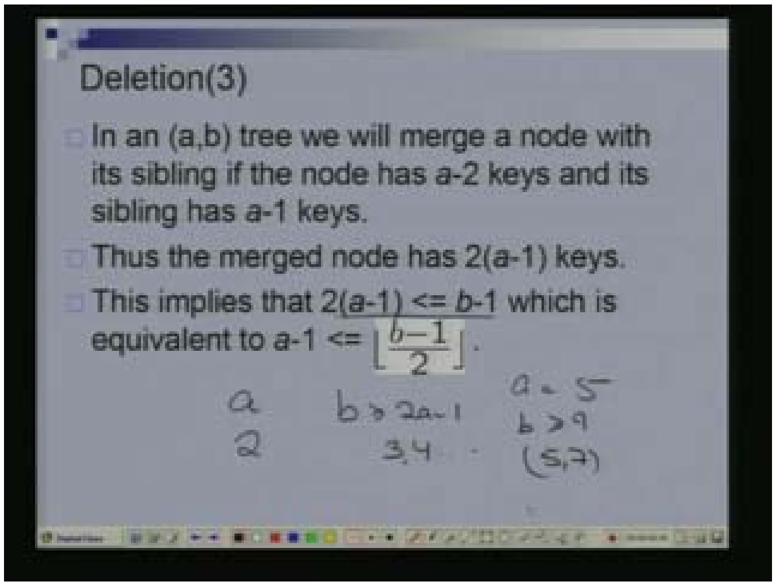
The minimum number of keys in a node is a-1, which means it had a-2 keys. I am trying to borrow one from its sibling but I am merging with the sibling which means that even the sibling does not have anything to lend me. Which means how much does the sibling have? The minimum number a-1.

If I am merging the node with the sibling then that means the sibling has a-1keys and the node itself has a-2keys. After merging the new node that gets created, how many key will it have? The sum of these two (a-2 and a-1) plus one. Why plus one? It will have 2a-1keys. Now 2(a=1) better be less than b-1. This is the same as saying that a-1 is better be less than $\frac{b-1}{2}$.

Why have I set flow? The a-1has to be an integer. As $a-1 <= \left\lfloor \frac{b-1}{2} \right\rfloor$and since a-1 is an integer it has to be less than or equal to the floor of $\frac{b-1}{2}$, floor means rounded down. This symbol means, if this is not an integer round it down to the nearest integer.

This ($a-1 <= \left\lfloor \frac{b-1}{2} \right\rfloor$) is the property that your a and b should satisfied. Let us just quickly see what this just means. Given the particular value of a, what are the different values b can take? You just have to look at this (2(a-1) <=b-1) again. So b can take a value 2a-1 or more. The b should be greater than or equal to 2a-1. When a was 2, what are the values b can take? 3, 4 and so on. That is the thing that we need to keep in mind. If we have a as 5, then b have to be at least 9. So you cannot have a (5, 7) tree, this will not work where (5, 9) tree is okay.

(Refer Slide Time: 45:36)



Deletion(3)

- In an (a,b) tree we will merge a node with its sibling if the node has a-2 keys and its sibling has a-1 keys.
- Thus the merged node has 2(a-1) keys.
- This implies that $\underline{2(a-1) <= b-1}$ which is equivalent to $a-1 <= \left\lfloor \frac{b-1}{2} \right\rfloor$.

a $\quad b \geq 2a-1$ $\quad a = 5$

2 $\quad 3,4$ $\quad b \geq 9$

$(5,9)$

We will see a quick summary. For insertion and deletion, we saw insertion and deletion in a-b trees. The height of an a-b tree we saw is log n and so insertion and deletion both take order log n time. The reason for that is the same as in the case of a 2-4 tree.

The reason why the insertion and deletion was taking log n time in 2-4 tree was the height was log n and we might have done some number of operations. But we were doing the operation proportional to the height. We might move up all the way, so first we move down in the case of insertion and deletion so that height is order log n. Then sometimes you are borrowing in the case of insertion, sometimes splitting, but the number of times you have to split is at most the height. In the case of deletion we would either borrow the key or we would merge or again the number of times we would have to do this is proportional to the height because every time we did one of these operation, we moved one level up or we just stop the entire process. So both of these operation take order log n time.

The other thing we saw was, what should be the relation between a and b for this to work. That is as far as this a-b tree was concern. The other thing we did today was red black tree and for that we saw the process of insertion. We saw that it takes only some number of re coloring and one rotation to complete an insertion. That is the key thing about the red black tree, they only require one rotation. Some number of re coloring or might have to re color many nodes but only one rotation is required and this is what gives the power. This is what makes them very fast in practice and they are faster then AVL trees for this reason. So the next class we are going to see the particular kind of the a-b tree which is called b-tree and what role does it plays specially in searching very large databases. That is what we are going to do in the next class.